目录

[实现自定义控件的三种方式 2](#_Toc12332)

[onMeasure三种测量方式以及和子控件的关系 2](#_Toc5479)

[Activity、PhoneWindow、DecorView、ViewRoot 3](#_Toc26064)

[WindowManager & WindowManagerService 4](#_Toc31611)

[Window 分类 4](#_Toc20530)

[WindowManagerService 5](#_Toc20885)

[DecorView 顶级View 6](#_Toc19654)

[ViewRoot 连接器 6](#_Toc9041)

[Activity & PhoneWindow & DecorView & ViewRoot 联系 7](#_Toc13516)

[DecorView 的创建 & 显示（View 绘制前准备） 7](#_Toc14523)

[Activity页面加载流程 9](#_Toc23736)

[（五）Android View 绘制流程 11](#_Toc5775)

[1、Measure 11](#_Toc6163)

[2、Layout 16](#_Toc25886)

[3、Draw 18](#_Toc11558)

[Android 三种动画 (帧动画 、补间动画、属性动画) 21](#_Toc7110)

[1、帧动画 21](#_Toc16169)

[2、补间动画 22](#_Toc31980)

[3、属性动画 23](#_Toc10962)

[View的刷新机制 24](#_Toc3779)

# 实现自定义控件的三种方式

Android实现自定义控件的方式一般有三种，第一种是继承现有的控件，重写相应的方法来扩展该控件的功能；

第二种是通过组合控件来实现自定义控件，譬如说可以自定义一个控件，让它既有ToggleButton的功能，又有ImageView的功能。

第三种是继承View类或者ViewGroup类，通过onDraw方法绘制控制，然后通过onMeasure方法设置控件的显现，最后为自己绘制的控件设置监听；

# onMeasure三种测量方式以及和子控件的关系

SpecMode有三种值，如下所示。

) UNSPECIFIED

父容器不对View有任何限制，要多大给多大，这种情况一般用于系统内部， 我们在开发过程中基本用不到。

）EXACTLY

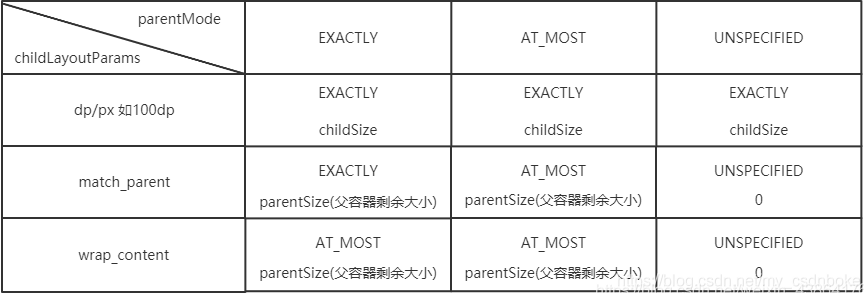
父容器已经检测出View所需要的精确大小，这个时候View的最终大小就是SpecSize所指定的值。它对应于LayoutParams中的match\_parent和具体的数值这两种模式。

）AT\_MOST

父容器指定了一个可用大小即SpecSize，View的大小不能大于这个值，具体是什么值要看不同View的具体实现。它对应于LayoutParams中的wrap\_content。

当嵌套子控件的时候：

通过父ViewGroup的测量规则和子View的布局参数，生成子View的测量规则，最后传递到子View的onMeasure()方法供自View进行自身大小的测量。



横轴：父的测量模式

纵轴：子的LayoutParams

# Activity、PhoneWindow、DecorView、ViewRoot

Activity 控制器

Activity并不负责视图控制，它只是控制生命周期和处理事件。真正控制视图的是Window。一个Activity包含了一个Window，Window才是真正代表一个窗口。Activity就像一个控制器，统筹视图的添加与显示，以及通过其他回调方法，来与Window、以及View进行交互。

Window 承载器

Window是视图的承载器，承载视图View的显示。内部持有一个 DecorView，而这个DecorView才是 view 的根布局。

Window是一个抽象类，实际在Activity中持有的是其子类PhoneWindow。PhoneWindow中有个内部类DecorView，通过创建DecorView来加载Activity中设置的布局R.layout.activity\_main。

Window 通过WindowManager将DecorView加载其中，并将DecorView交给ViewRoot，进行视图绘制以及其他交互。

# WindowManager & WindowManagerService

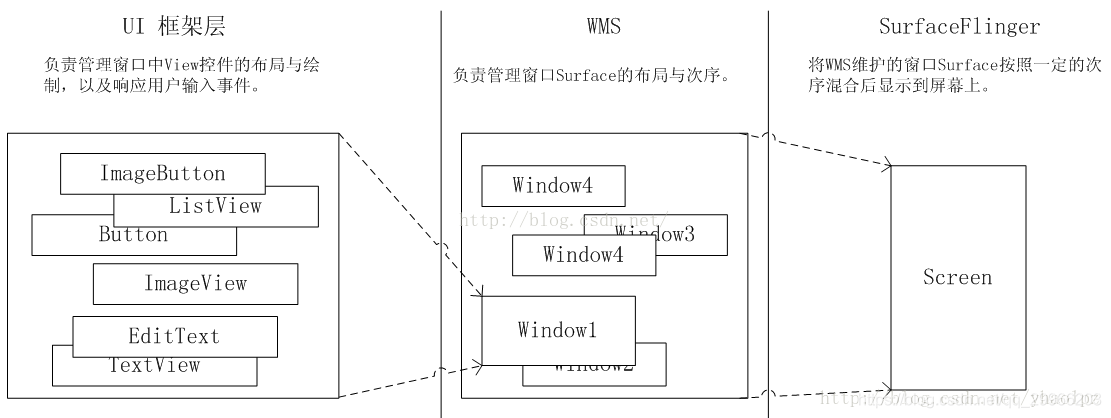
## Window 分类

Window 有三种类型，分别是应用 Window、子 Window 和系统 Window。应用类 Window 对应一个 Acitivity，子 Window 不能单独存在，需要依附在特定的父 Window 中，比如常见的一些 Dialog 就是一个子 Window。系统 Window是需要声明权限才能创建的 Window，比如 Toast 和系统状态栏都是系统 Window。

Window 是分层的，每个 Window 都有对应的 z-ordered，层级大的会覆盖在层级小的 Window 上面，这和 HTML 中的 z-index 概念是完全一致的。在三种 Window 中，应用 Window 层级范围是 1 ~ 99，子 Window 层级范围是 1000 ~ 1999，系统 Window 层级范围是 2000 ~ 2999

## WindowManagerService

WindowManagerService 就是位于 Framework 层（Android Application层）的窗口管理服务，它的职责就是管理系统中的所有窗口。窗口的本质是什么呢？其实就是一块显示区域，在 Android 中就是绘制的画布：Surface，当一块 Surface 显示在屏幕上时，就是用户所看到的窗口了。WindowManagerService 添加一个窗口的过程，其实就是 WindowManagerService 为其分配一块 Surface 的过程，一块块的 Surface 在 WindowManagerService 的管理下有序的排列在屏幕上，Android 才得以呈现出多姿多彩的界面。于是根据对 Surface 的操作类型可以将 Android 的显示系统分为三个层次，如下图：



一般的开发过程中，我们操作的是 UI 框架层，对 Window 的操作通过 WindowManager 即可完成，而 WindowManagerService 作为系统级服务运行在一个单独的进程，所以 WindowManager 和 WindowManagerService 的交互是一个 IPC 过程。

* WindowManager  
  在实际使用中无法直接访问 Window，我们对 Window 的操作是通过 WindowManager 来完成的，WindowManager 是一个接口，它继承自只有三个方法的 ViewManager 接口：

public interface ViewManager{

public void addView(View view, ViewGroup.LayoutParams params);

public void updateViewLayout(View view, ViewGroup.LayoutParams params);

public void removeView(View view);

}

这三个方法其实就是 WindowManager 对外提供的主要功能，即添加 View、更新 View 和删除 View。WindowManager 最终都会通过一个 IPC 过程将操作移交给 WindowManagerService 这个位于 Framework 层的窗口管理服务来处理。

## DecorView 顶级View

DecorView是FrameLayout的子类，它可以被认为是Android视图树的根节点/顶级视图。用于显示 & 加载视图。它内部包含一个竖直方向的LinearLayout，在这个LinearLayout里面有两个部分：标题栏(根据Theme设置，有的布局没有)，下面的是内容栏。 具体情况和Android版本及主体有关，在Activity中通过setContentView所设置的布局文件其实就是被加到内容栏之中的，成为其唯一子View，在代码中可以通过content来得到对应加载的布局。

ViewGroup content = (ViewGroup)findViewById(android.R.id.content);

ViewGroup rootView = (ViewGroup) content.getChildAt(0);

## ViewRoot 连接器

所有View的绘制以及事件分发等交互都是通过ViewRoot来执行或传递的。

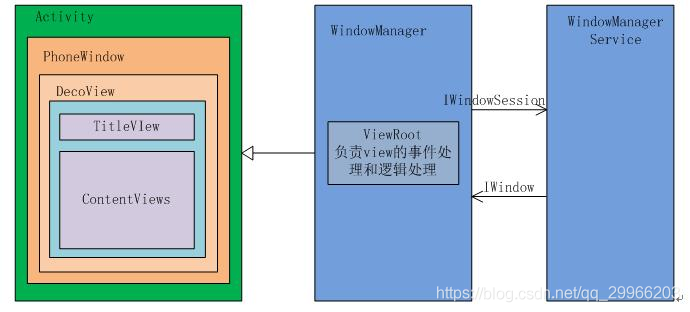
ViewRoot对应ViewRootImpl类。它的作用包括：

（1）连接WindowManager 和 DecorView

（2）完成View的绘制流程

即ViewRoot可以与WMS交互通讯，调整窗口大小及分布；可以接收事件并向DecorView分发，Android的所有触屏事件、按键事件、界面刷新等事件都是通过ViewRoot进行分发的；可以完成View的三大绘制流程：测量、布局和绘制。

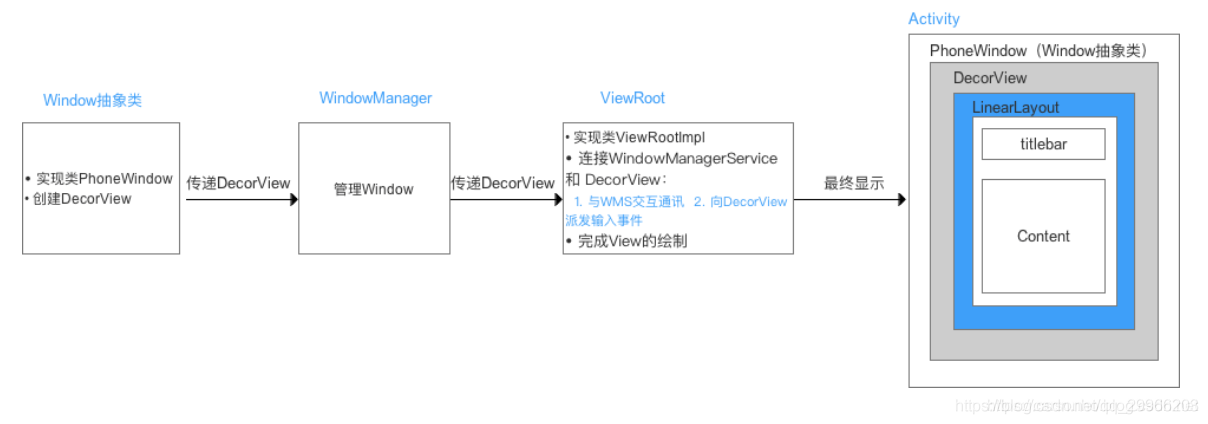
## Activity & PhoneWindow & DecorView & ViewRoot 联系



Activity就像个控制器，不负责视图部分。Window像个承载器，装着内部视图。DecorView就是个顶层视图，是所有View的最外层布局。ViewRoot像个连接器，负责沟通，通过硬件的感知来通知视图，进行用户之间的交互。

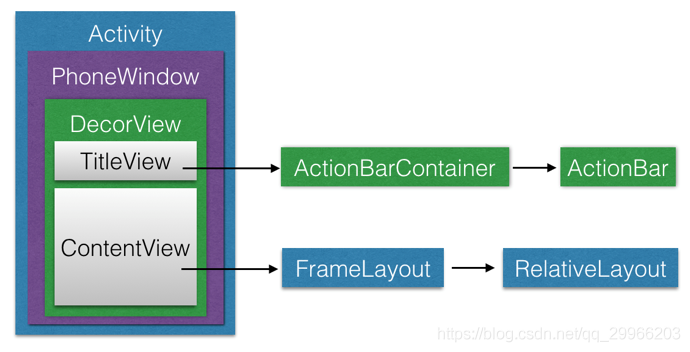
## DecorView 的创建 & 显示（View 绘制前准备）

工作流程



源码见https://blog.csdn.net/qq\_29966203/article/details/93157511

# Activity页面加载流程



### **（1） Window**

Android中Window的具体实现（也是唯一实现）——PhoneWindow，平时调用setContentView()方法设置Activity的用户界面时，实际上就完成了对所关联的PhoneWindow的ViewTree的设置，可以通过Activity类的requestWindowFeature()方法来定制Activity关联PhoneWindow的外观，这个方法实际上做的是把我们所请求的窗口外观特性存储到了PhoneWindow的mFeatures成员中，在窗口绘制阶段生成外观模板时，会根据mFeatures的值绘制特定外观。

### **（2） setContentView（）**

这个方法只是完成了Activity的ContentView的创建，而并没有执行View的绘制流程。

public void setContentView(@LayoutRes int layoutResID) {

getWindow().setContentView(layoutResID);

. . .

}

getWindow()方法会返回Activity所关联的PhoneWindow，也就是说，实际上调用到了PhoneWindow的setContentView()方法

### **LayoutInflater.inflate（）**

### **View绘制的起点**

View的绘制是由ViewRoot来负责的，每个应用程序窗口的decorView都有一个与之关联的ViewRoot对象，这种关联关系是由WindowManager来维护的。

decorView与ViewRoot的关联关系是在什么时候建立的呢？答案是Activity启动时，ActivityThread.handleResumeActivity()方法中建立了它们两者的关联关系

ViewRoot完成View绘制的过程：

建立好了decorView与ViewRoot的关联后，ViewRoot类的requestLayout()方法会被调用，以完成应用程序用户界面的初次布局。

实际被调用的是ViewRootImpl类的requestLayout()方法

源码如下：

@Override

public void requestLayout() {

if (!mHandlingLayoutInLayoutRequest) {

// 检查发起布局请求的线程是否为主线程

checkThread();

mLayoutRequested = true;

scheduleTraversals();

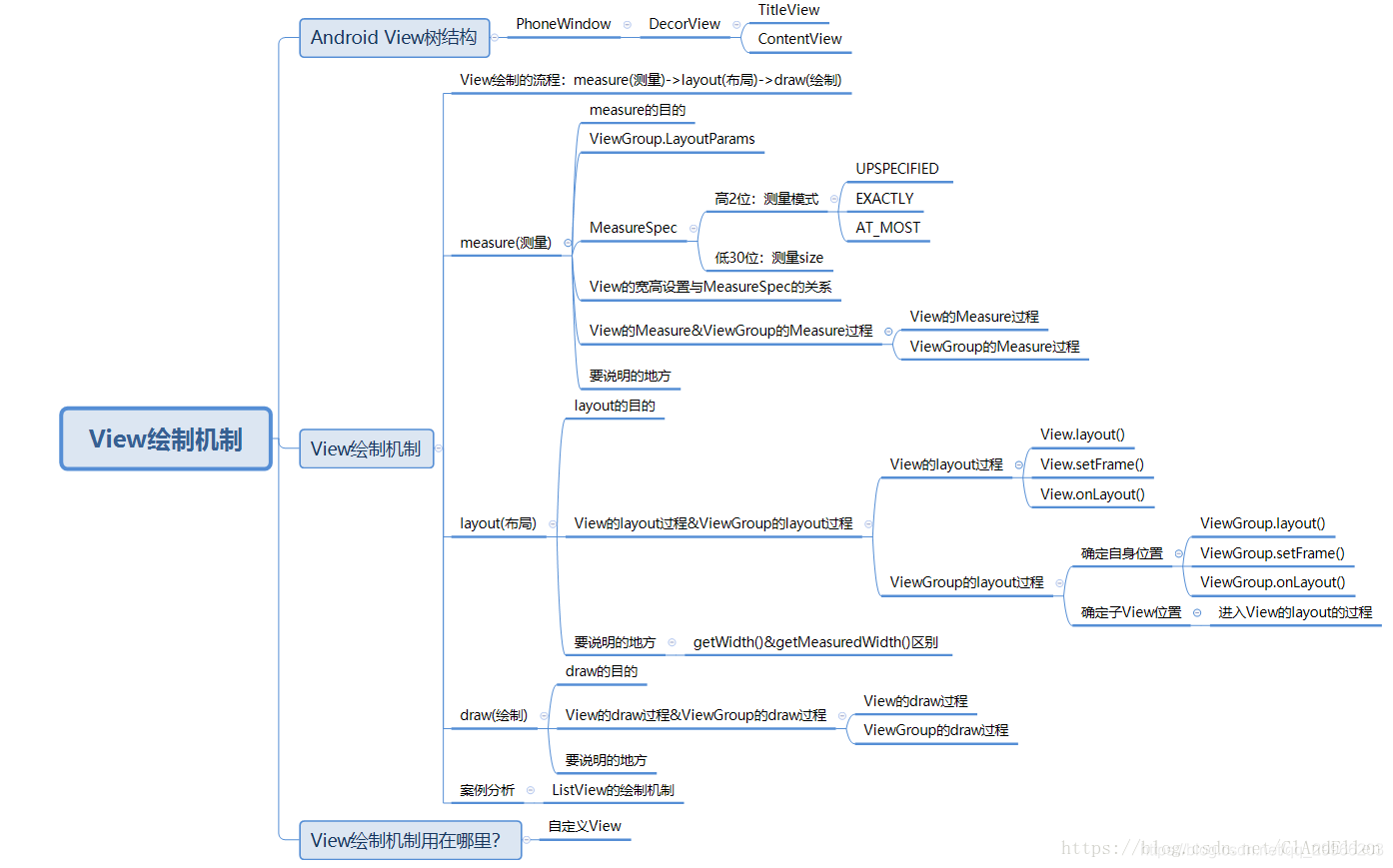
}

}

scheduleTraversals()方法来调度一次完成的绘制流程，该方法会向主线程发送一个“遍历”消息，最终会导致ViewRootImpl的performTraversals()方法被调用。

以performTraversals()为起点，来分析View的整个绘制流程。对遍历的每个View进行三个阶段的绘制：measure、draw、layout

# （五）Android View 绘制流程

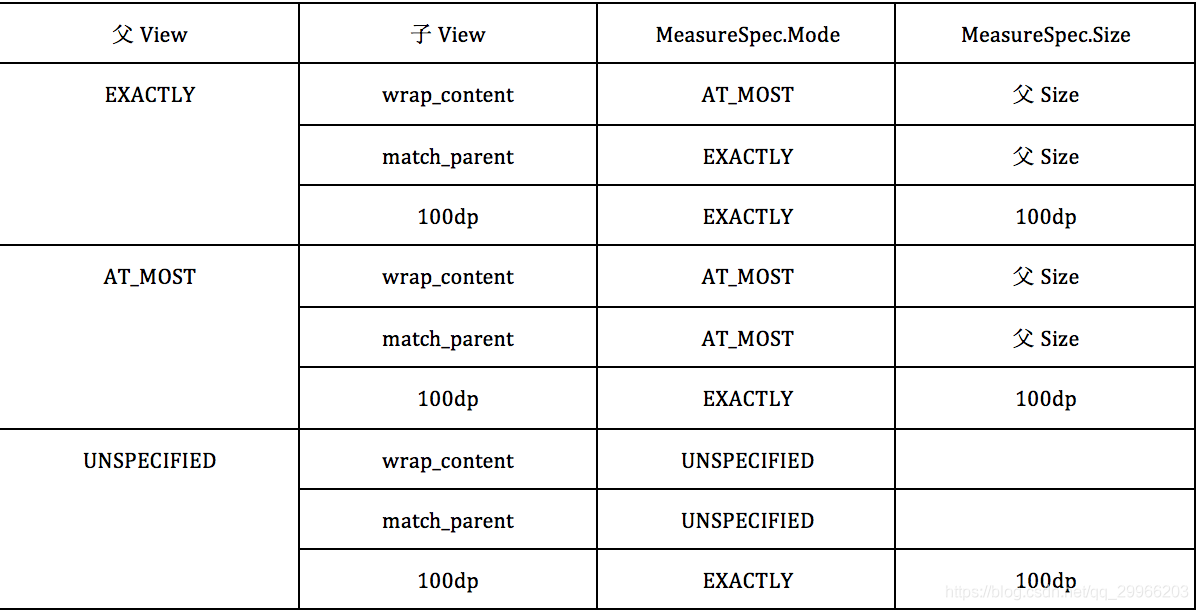


# 1、Measure

Measure的目的就是测量View的宽和高

#### **（1）MeasureSpec理解——父容器传递给子容器的布局要求**

由父View的MeasureSpec和子View的LayoutParams通过简单的计算得出一个针对子View的测量要求（测量模式+测量参数）。对于一个ViewGroup或者View的宽高而言，都一一对应一个MeasureSpec。



****ViewGroup.LayoutParams****

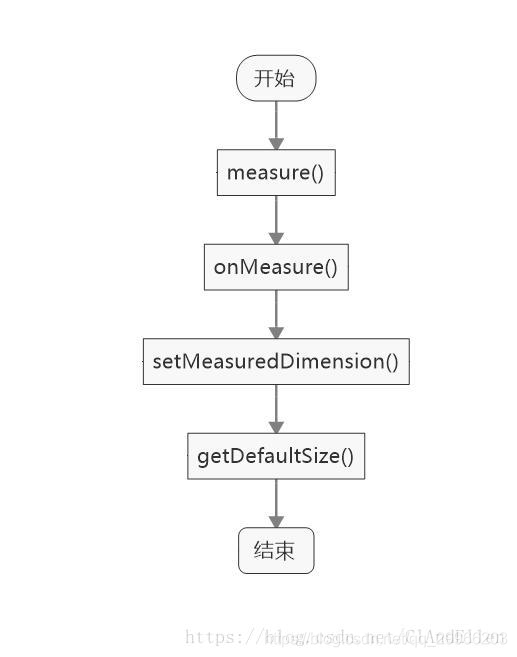
LayoutParams类就是指定View宽高等布局参数而被使用的。

具体值：以px或者dp为单位

match \_ parent：特性和fill\_parent相似，Android版本大于2.3使用

wrap \_ content：自适应大小，强制性地使视图扩展以便显示其全部内容(含 padding )

#### **（2）View的Measure过程（默认）**



measure():基本测量逻辑的判断。

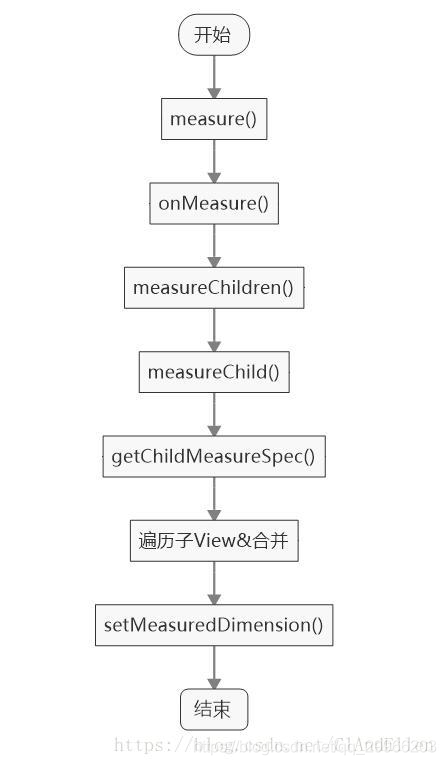
onMeasure():调用下面两个

setMeasuredDimension():存储测量后的宽和高。  
getDefaultSize():根据View宽/高的测量规格计算View的宽/高。

View默认是测量很简单，大部分情况就是拿计算出来的MeasureSpec的size 当做最终测量的大小。

对于其他的一些View的派生类，如TextView、Button、ImageView等，它们的onMeasure方法系统了都做了重写，一般先去测量字符或者图片的高度等，然后拿到View本身content这个高度（字符高度等），如果MeasureSpec是AT\_MOST，而且View本身content的高度不超出MeasureSpec的size，那么可以直接用View本身content的高度（字符高度等），而不是像View.java 直接用MeasureSpec的size做为View的大小。

#### **（3）ViewGroup的Measure过程**



measure():基本测量逻辑的判断。

onMeasure():调用measureChildren。

measureChildren():遍历子View并对子View进行测量，后续会调用measureChild()方法。

measureChild():计算出单个子View的MeasureSpec，通过调用getChildMeasureSpce()方法实现，调用每个子View的measure()方法进行测量。

getChildMeasureSpec():计算出子View的MeasureSpec。

setMeasuredDimension():存储测量后的宽和高。

##### **4、布局流程**

View绘制的起点

View的绘制是由ViewRoot来负责的。每个应用程序窗口的decorView都有一个与之关联的ViewRoot对象，这种关联关系是由WindowManager来维护的（Activity启动时，ActivityThread.handleResumeActivity()方法中建立了它们两者的关联关系。）

当建立好了decorView与ViewRoot的关联后，ViewRoot类的requestLayout()方法会被调用，以完成应用程序用户界面的初次布局。实际被调用的是ViewRootImpl类的requestLayout()方法

这个方法的源码如下：

@Override

public void requestLayout() {

if (!mHandlingLayoutInLayoutRequest) {

// 检查发起布局请求的线程是否为主线程

checkThread();

mLayoutRequested = true;

scheduleTraversals();

}

}

上面的方法中调用了scheduleTraversals()方法来调度一次完成的绘制流程，该方法会向主线程发送一个“遍历”消息，最终会导致ViewRootImpl的performTraversals()方法被调用。下面，我们以performTraversals()为起点，来分析View的整个绘制流程。

###### **step1.DecorView(FragmentLayout)——整个View的ROOT**

绘制入口是由ViewRootImpl的perform Traversals（）发起Measure,Layout,Draw等流程

那么接下来在FrameLayout 的onMeasure()方法DecorView开始for循环测量自己的子View,测量完所有的子View再来测量自己，根据View树知接下来要测量ViewRoot的大小

###### **step2.ViewRoot(LinearLayout)**

ViewRoot 的MeasureSpec mode应该等于EXACTLY（DecorView MeasureSpec 的mode是EXACTLY，ViewRoot的layoutparams 是match\_parent），size 等于DecorView的size

ViewRoot是一个LinearLayout ，ViewRoot.measure最终会执行的LinearLayout 的onMeasure 方法，LinearLayout 的onMeasure 方法又开始逐个测量它的子View（measureChildWithMargins），那么根据View的层级图，接下来测量的是header（ViewStub）,由于header的Gone，所以直接跳过不做测量工作，所以接下来轮到ViewRoot的第二个child content（android.R.id.content）

###### **step3.Content(LinearLayout)**

由于ViewRoot 的mPaddingBottom=100px(id/statusBarBackground的View的高度刚好等于100px，所以计算出来Content（android.R.id.content） 的MeasureSpec 的高度少了100px )它的宽高的mode 根据算出来也是EXACTLY（ViewRoot 是EXACTLY和android.R.id.content 是match\_parent）。

Content（android.R.id.content） 是FrameLayout，递归调用开始准备计算id/linear的MeasureSpec

###### **step4.linear(LinearLayout)**

id/linear的heightMeasureSpec 的mode=AT\_MOST，因为id/linear 的LayoutParams 的layout\_height=“wrap\_content”，由于id/linear 的 android:layout\_marginTop=“50dp” 使得lp.topMargin=200px (本设备的density=4，px=4\*pd)，在计算后id/linear的heightMeasureSpec 的size 少了200px。

###### **step5.text(TextView)**

算出id/text 的MeasureSpec 后TextView 拿着刚才计算出来的MeasureSpec（mode=AT\_MOST,size=1980）,这个就是对TextView的高度和宽度的约束，进到TextView 的onMeasure(widthMeasureSpec,heightMeasureSpec) 方法

TextView字符的高度（也就是TextView的content高度[wrap\_content]）测出来=107px，107px 并没有超过1980px(允许的最大高度)，所以实际测量出来TextView的高度是107px。

最终算出id/text 的mMeasureWidth=1440px,mMeasureHeight=107px。

###### **step6 view(View)**

id/linear 的子View的高度都计算完毕了，接下来id/linear就通过所有子View的测量结果计算自己的高宽，id/linear是LinearLayout，所有它的高度计算简单理解就是子View的高度的累积+自己的Padding.

最终算出id/linear出来后，id/content 就要根据它唯一的子View id/linear 的测量结果和自己的之前算出的MeasureSpec一起来测量自己的结果，具体计算的逻辑去看FrameLayout onMeasure 函数的计算过程。以此类推，接下来测量ViewRoot,然后再测量id/statusBarBackground,最后测量DecorView 的高宽，最终整个测量过程结束。

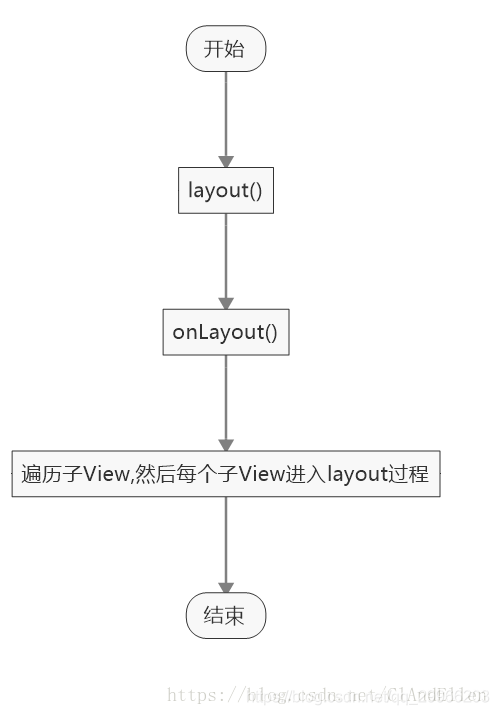
# 2、Layout

layout的主要作用 ：根据子视图的大小以及布局参数将View树放到合适的位置上。确认View&ViewGroup的四个顶点的位置（从而确定位置）,left,top,right,bottom

#### **2.入口DecorView**

mView.layout(0, 0, mView.getMeasuredWidth(), mView.getMeasuredHeight());

#### **3.ViewGroup的layout函数**



layout():调用layout()方法计算ViewGroup自身的位置，在此方法调用路径中有一个方法特别重要，这个方法就是setFrame()，它的作用就是根据传入的4个位置值，设置View本身的四个顶点位置，也就是用来确定最终View的位置的。接下来就是回调onLayout()方法。

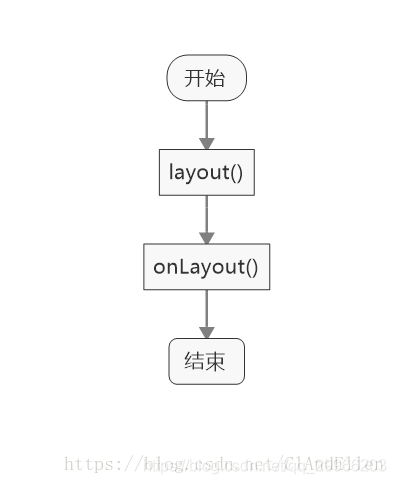
onLayout():对于ViewGroup而言，它不仅仅要确认自身的位置，它还要计算它的子View的位置，因此onLayout的作用就是遍历并计算每个子View的位置。

遍历自己的孩子，然后调用child.layout(l, t, r, b) ，给子view 通过setFrame(l, t, r, b) 确定位置，

而重点是(l, t, r, b) 怎么计算出来的呢。是通过之前measure测量出来的MeasuredWidth和MeasuredHeight、在xml 设置的Gravity、RelativeLayout 的其他参数等等一起来确定子View在父视图的具体位置的。

具体的计算过程不同的ViewGroup 的实现都不一样（FragmentLayout\RelativeLayout\LinearLayout）

#### **4.View的layout函数**

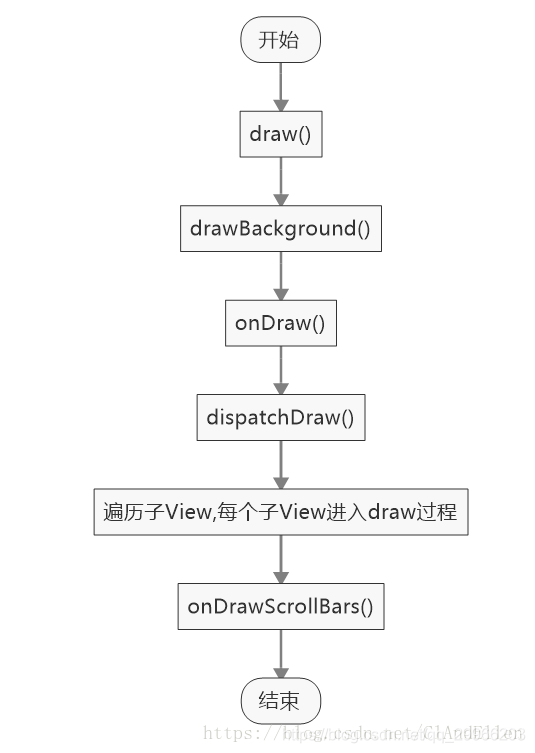


layout():调用layout()方法主要为了计算View自身的位置，在此方法调用路径中有一个方法特别重要，这个方法就是setFrame()，它的作用就是根据传入的4个位置值，设置View本身的四个顶点位置，也就是用来确定最终View的位置的。接下来就是回调onLayout()方法。

onLayout():对于View的onLayout()方法来说，它是一个空实现。为什么View的onLayout()方法是空实现呢？因为onLayout()方法作用是计算此VIew的子View的位置，对于单一的View而言，它并不存在子View，因此它肯定是空实现啦！

# 3、Draw

#### **2、对ViewGroup绘制**



draw():绘制ViewGroup自身。

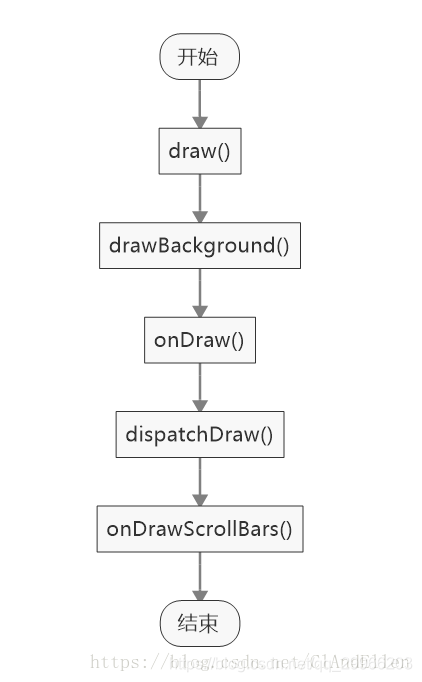
drawBackground():绘制ViewGroup自身的背景。

onDraw():绘制View自身的内容。

dispatchDraw():对于ViewGroup而言，它是存在子View的，因此此方法就是用来遍历子View，然后让每个子View进入Draw过程从而完成绘制过程。

onDrawScrollBars():ViewGroup的装饰绘制。

#### **3、对View绘制**



draw():绘制View自身。

drawBackground():绘制View自身的背景。

onDraw():绘制View自身的内容。

dispatchDraw():对于View而言，它是空实现，因为它的作用是绘制子View的，因为单一的View没有子View，因此它是空实现。

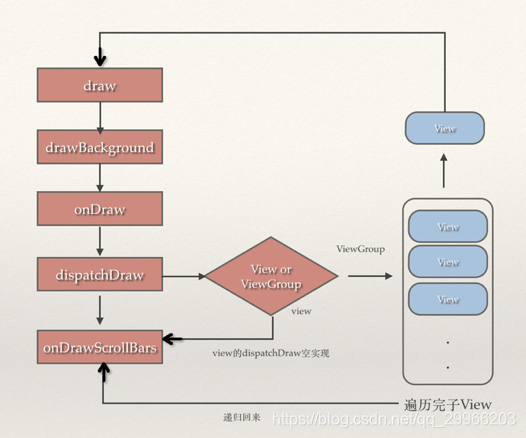
onDrawScrollBars():从名字可以看出，它是绘制滑动条等装饰的，比如ListView的滑动条。

onDraw(canvas) 方法是view用来draw 自己的，具体如何绘制，颜色线条什么样式就需要子View自己去实现，View.java 的onDraw(canvas) 是空实现，ViewGroup 也没有实现，每个View的内容是各不相同的，所以需要由子类去实现具体逻辑。

#### **4、对当前父View的所有子View绘制**

就是遍历子View然后drawChild(),drawChild()方法实际调用的是子View.draw()方法,ViewGroup类已经为我们实现绘制子View的默认过程

#### **5、对View滚动条绘制**



# Android 三种动画 (帧动画 、补间动画、属性动画)

## 1、帧动画

帧动画是依次展示n张静态图片，造成动画的错觉，类似看视频一样。

使用方式

在drawable目录下定义XML文件，根节点为animation-list，然后放入定义更好的图片

oneshot 表示是否重复播放动画

使用 AnimationDrawable 展示动画

AnimationDrawable animationDrawable = (AnimationDrawable) button.getBackground();

animationDrawable.start();//开启动画

## 2、补间动画

补间动画也称view动画，补间动画只需指定动画开始，以及动画结束"关键帧"， 而动画变化的"中间帧"则由系统计算并补齐。主要有：

AlphaAnimation：透明度渐变效果，创建时许指定开始以及结束透明度，还有动画的持续 时间，透明度的变化范围(0,1)，0是完全透明，1是完全不透明；对应<alpha/>标签！

ScaleAnimation：缩放渐变效果，创建时需指定开始以及结束的缩放比，以及缩放参考点， 还有动画的持续时间；对应<scale/>标签！

TranslateAnimation：位移渐变效果，创建时指定起始以及结束位置，并指定动画的持续 时间即可；对应<translate/>标签！

RotateAnimation：旋转渐变效果，创建时指定动画起始以及结束的旋转角度，以及动画 持续时间和旋转的轴心；对应<rotate/>标签

AnimationSet：组合渐变，就是前面多种渐变的组合，对应<set/>标签

animation = AnimationUtils.loadAnimation(this,R.anim.anim\_set);

img.startAnimation(animation);

动画也可以设置监听setAnimationListener(new AnimationListener())方法

onAnimationStart()：动画开始

onAnimtaionRepeat()：动画重复

onAnimationEnd()：动画结束

过场动画

Fragment： 我们可以调用FragmentTransaction对象的setTransition(int transit) 为Fragment指定标准的过场动画

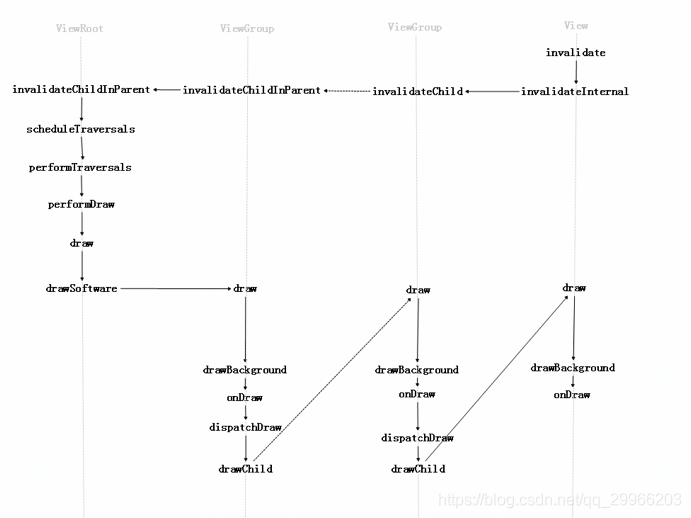
Activty设置过场动画非常简单，调用的方法是：overridePendingTransition(int enterAnim, int exitAnim)，在startActivity(intent)或者finish()后添加

## 3、属性动画

属性动画可以对任何对象的属性做动画而不仅仅是View，甚至可以没有对象。除了作用对象进行扩展外，属性动画的效果也加强了，不仅能实现View动画的4中效果，还能实现其它多种效果，这些效果都是通过ValuAnimator或ObjectAnimator、AnimatorSet等来实现的。

# View的刷新机制

### **1、invalidate（）流程图**



invalidate主要给需要重绘的视图添加DIRTY标记，并通过不断回溯父视图做矩形运算求得真正需要绘制的区域，并最终保存在ViewRoot中的mDirty变量中，最后调用scheduleTraversals发起重绘请求，scheduleTraversals会发送一个异步消息，最终调用performTraversals()执行重绘（performTraversals（）遍历所有相关联的 View ，触发它们的 onDraw 方法进行绘制）

### **2、源码分析**

1、子View需要刷新时，调用invalidate，通知父View完成——首先找到自己父View(View的成员变量mParent记录自己的父View)，然后将AttachInfo中保存的信息告诉父View刷新自己。

2、在invalidate中，调用父View的invalidateChild，这是一个从第向上回溯的过程，每一层的父View都将自己的显示区域与传入的刷新Rect做交集。

3、向上回溯的过程直到ViewRoot那里结束，由ViewRoot对这个最终的刷新区域重新绘制performTraversals。

### **3、invalidate和postInvalidate**

1、Invalidate()方法不能放在线程中，所以需要把Invalidate()方法放在Handler中。在MyThread中只需要在规定时间内发送一个Message给handler，当Handler接收到消息就调用Invalidate()方法。postInvalidate()方法就可以放在线程中做处理，就不需要Handler（postInvalidate 最终通过 Handler 切换到主线程，调用 invalidate）

2、Invalidate()方法和postInvalidate()都可以在主线程中调用而刷新视图。